Fused sparse multi-head-self-attention

AHAN GUPTA, GANGMUK LIM, and SELIN YILDIRIM

Multi-head-self-attention (MHSA) mechanisms achieve state-of-the-art (SOTA) performance across natural language processing and vision tasks. Unfortunately, their quadratic dependence on sequence lengths has bottlenecked inference speeds. To circumvent this bottleneck, researchers have explored two orthogonal directions. (1) Using sparse-MHSA models, where a subset of full-attention is computed. (2) IO-aware fusions of MHSA that reduce the number of reads and writes to high-bandwith-memory (HBM) like in flash-attention [8].

We investigate CUDA kernel level optimisations at the confluence of these two methods: fused sparse-multi-head-self-attention (FSMHSA). Our findings are two-fold. (1) FSMHSA is a combination of three kernels: sampled-dense-dense-matrix multiplication (SDDMM), softmax, sparse-matrix-multiplication (SpMM) ultimately outputting a dense matrix. FSMHSA kernels therefore do not require sparse data formats to represent sparsity in inputs. (2) GPU-tiling - where to place thread-blocks in the output (and intermediary) matrices - is the key optimisation to exploit sparsity, enhance locality and reduce thread-divergence.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Ahan Gupta, Gangmuk Lim, and Selin Yildirim. 2018. Fused sparse multi-head-self-attention. *J. ACM* 37, 4, Article 111 (August 2018), 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Multi-head-self attention (MHSA) [16] based models have been adopted in a variety of domains. They are the cornerstone of state-of-the-art (SOTA) trans- former architectures like GPT-4 [11] and Claude [3]. Despite their prevalence, training large transformers requires immense resources due to the quadratic de- pendence on input sequence lengths. This is exacerbated by increasingly larger models (e.g. GPT-3 xl - 175B parameters [5]) and pre-training data-set sizes (e.g. C4 750 GB [13]).

To circumvent this quadratic dependence, researchers have proposed to sparsify the self-attention computation by computing a subset of the full attention matrix [4, 7, 10, 12]. A variety of such sparse-MHSA patterns have been proposed. These models are both highly accurate and fast, linearizing the quadratic self-attention computation [4, 14]. Moreover, they demonstrate immense utility in fixed-time-budget training settings, where sparse-MHSA models train on more data compared to full-attention based models, consequently yielding models of lower perplexity [9]. In spite of their promise, implementing high-performant sparse-MHSA in modern deep learning compilers [6, 15] and vendor-libraries [1, 2] has been a challenge. New research indicates that code-generation frameworks that rely on the regularity of sparsity and regularly sparse data-structures like the affine- compressed-sparse-row format yield considerable generality and performance across a variety of sparse-MHSA patterns.

Authors' Contact Information: Ahan Gupta; Gangmuk Lim; Selin Yildirim.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Orthogonally, promising new compiler research has explored fusing kernels across all the steps of MHSA like in FlashAttention [8]. However, these fusions are hand-written and tuned for specific patterns and are cumbersome to implement, lacking generality.

We investigate optimisations at the confluence of the two methods: fused sparse-multi-head-self-attention (FSMHSA). Nevertheless, generating high-performance fused code for a variety of sparse-patterns is non-trivial due to two reasons. (1) Uncovering a parallelisation strategy detailing the computation of each thread-block is complex as intermediate tensors differ in size. An incorrect parallelization strategy will result in thread-blocks either doing excess or no work. (2) Sparse tensor algebra induces complex thread-access patterns resulting in un-coalesced memory accesses, thread-divergence and low levels of locality within threads of a thread-block. Generating good fused code for sparse tensor kernels inherits both these issues and is exceedingly complex.

We uncover several findings through this research. (1) Conventional sparse kernels require sparse formats to store sparse inputs, however FSMHSA kernels both ingest and produce dense outputs, resulting in no need to leverage sparse formats to represent intermediate sparse tensors. (2) We can exploit the sparsity in FSMHSA through thread-block placement, controlling the computation and the inputs thread-blocks read and write to exploit the sparsity and reduce redundant compute in FSMHSA.

2 BACKGROUND AND MOTIVATION

Sparse Attention To alleviate the quadratic computation in self-attention. Researchers have proposed a variety of *sparsification* techniques to reduce the size and memory of computing A_i. These techniques compute some subset of the values of A_i controlled by a mask matrix M, reducing the runtime of MHSA [4, 7] by computing:

$$\underbrace{\underbrace{[softmax(\overbrace{M \otimes QW_i^Q(KW_i^K)^T}^{SDDMM})]}_{A_i^s} VW_i^V}_{SpMM} \tag{1}$$

where mask M is a mask of 0s and 1s and $\otimes$ is a pair-wise product. The product: $M \otimes (QW_i^Q(KW_i^K)^T)$ in traditional sparse computing terminology is a sampled dense dense matrix multiplications (SDDMM), whilst the product: $A_i^s VW_i^V$ is a sparse matrix dense matrix multiplication (SpMM). However, compared to the sparsity levels studied in sparse computing literature, M is both moderately sparse and regular.

Sparse-MHSA Patterns A variety of sparse transformers have been proposed in the literature [4, 7, 10**?**]. For example, the strided and windowed pattern (figure 1 left and middle) which implement Longformer (written in TVM) [4, 6], and the blocked pattern (figure 1 right) which implements Reformer (written in JAX), and sparse-transformer (written in triton) [7, 15**?**].

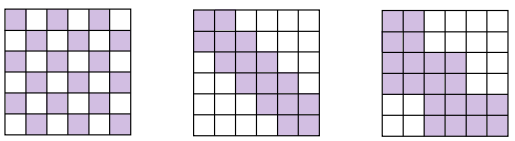

Fig. 1. Examples of 3 commonly occurring sparse-MHSA patterns in the literature. Strided (left figure), Windowed (middle figure) [4], Blocked (right figure) [7, 10]. Full attention computes all points

Flash-Attention Flash-attention [8] is an IO-aware fusion of MHSA that reduces the number of reads and writes to HBM. The parallelization strategy entails spawning one column of

thread-blocks along the attention matrix and sequentially aggregating the answer of the output matrix. For an extensive explanation of flash-attention, see [8].

3 PROBLEM ANALYSIS

We demonstrate the difficulty and complexity involved in (1) high-performance sparse tensor algebra kernel implementation and (2) fusion between disparate CUDA kernels.

Challenges in sparse tensor algebra kernels. Sparse tensor algebra kernels operate on sparse formats (such as CSR & COO). They are typically implemented with high-performance vendor libraries such as cuSPARSE, or triton [1, 2, 15]. Nevertheless, yielding benefits over dense computations are non-trivial. (1) Sparse data formats require complex nested memory indexing, resulting in random and uncoalesced memory access patterns. (2) Sparse data-structures prohibit high levels of re-use, as they are memory-bound: requiring data to be read for small amounts of compute. (3) Sparsity is typically random, resulting in threads within a warp deviating in control flow. Due to these reasons, cuSPARSE, a highly optimised vendor library to implement sparse primitives only yields benefits over dense baselines (such as cuBLAS) for extremely low sparsity levels. We demonstrate this in figure 2 where cuSPARSE looses performance to cuBLAS at density levels of 20% and greater.

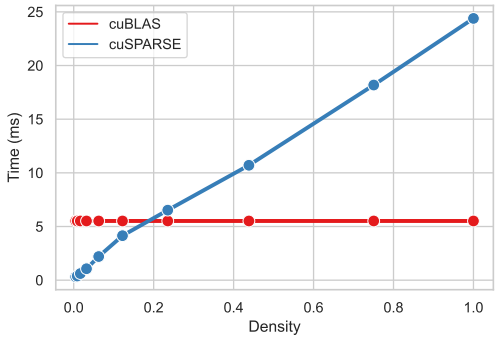

Fig. 2. A comparison of the SpMM primitive of cuSPARSE against the dense library cuBLAS.

Challenges in kernel fusion. Kernel fusion is a classic compiler optimisation to enhance locality by reducing reads/writes to memory. Nevertheless, innovating on a parallelization strategy for fused kernels is non-trivial on vector machines like GPUs. In the CUDA programming model, kernels are decomposed into threads with each thread computing one, or several, output values. However, a series of fused kernels produces intermediate tensors of differing sizes. Naive thread-block mappings will induce thread-blocks which either do excess work or little to no work, causing load imbalances and slow overall kernel time. This is the case in the fused-attention kernel, where the intermediate attention matrix and output matrix are of different sizes.

4 DESIGN

We designed a flash-attention kernel from scratch to prototype our ideas and optimisations over. We give an overview of the flash-attention implementation and preliminary optimizations.

Flash-attention kernel implementation. Our flash-attention implementation is given in listing 1. We parallelize over the inner loop, iterating each thread-block over the outer-loop (for a comprehensive overview of the algorithm see [8]). The kernel proceeds by first computing the $S_{i,j} = Q_i K_j^T$ product in a tiled fashion (lines 21-26). It then computes metadata over the $S_{i,j}$ such as row-sums, and computes $P_{i,j} = e^{S_{i,j}}$ (line 28). Finally, it uses this metadata, $P_{i,j}$, and Values matrices to compute a chunk of the output O_i (lines 30-38).

Thread-block j^{th} iteration specialisation. We observe that sparse patterns fall into two categories: (1) polygonal patterns, (2) strided patterns. Polygonal patterns are dense, with no gaps between neighboring points like the windowed and blocked pattern. Strided patterns have regular zero values. Polygonal patterns have areas of non-zero values clustered together. We can therefore

specialise the iteration start and end of each thread-block to the relevant *span* of non-zero values it should compute. Suppose we spawn thread-blocks of size $n \times m$, then the start and end-points of thread-block j will be:

$$[\min_{i \in [n]} Col\_idx[i + j * n], \max_{i \in [n]} Col\_idx[i + j * n]]$$

Where $Col\_idx[k]$ returns an array consisting of all the column indexes of the non-zero values in the k^{th} row. This is specific to each thread-block and guaranteed to *not* cause thread-divergence. Moreover, this can be computed just-in-time, at the point of reading a sparse mask and reused for every iteration of inference/training.

5 IMPLEMENTATION

```
__global__ void sparsefmha(float * queries, float * keys, float * values, float*
    answer, coord * start_end) {
      int tx = threadIdx.x;
      int bx = blockIdx.x;
      int idx = bx * blockDim.x + tx;

      // Shared memory variables. TBs are size bc by br.
      // See original flash-attention paper for further
      // clarification on variable names.

      // Sij -> intermediate attention value.
      __shared__ float sij[bc][br];
      // Oij -> intermediate output value.
      __shared__ float oij[bc][br];
      // Extra metadata, e.g. lij, mi, mij etc...
      __shared__ li[bc];
      ...

     // We parallelize over the inner loop and retain the outer loop.
     for (int j = start_end[idx].start; j < start_end[idx].end; j++) {
        // Compute Sij using tiling...
        for (int a = 0; a < ceil(hidden_dim/br); a++) {
            // collaboratively load keys and values.
            ...
            // Multiply and store into Sij.
            ...
        }

        // Compute metadata, e^Sij etc...

        // Finally we compute the output in a tiled fashion.
        for (int a = 0; a < ceil(hidden_dim/br); a++) {
            // Load collaboratively into Oij.
            ...
            // Use Sij and metadata to scale Oij accordingly
            ...
            // Write back to answer.
            ...
        }

     }

}
```

Listing 1. Flash-attention with span-specialisation

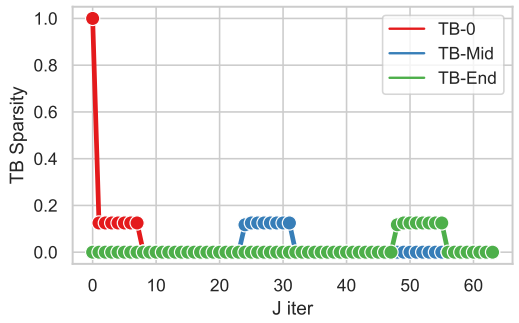

Fig. 3. TB Sparsity across the different iterations of j, the outer loop of flash-attention for the blocked pattern.

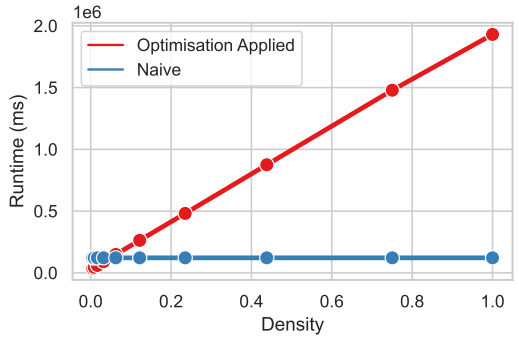

Fig. 4. Performance of sparsity-aware fused kernel

6 EXPERIMENTS

We conducted two sets of experiments. (1) We analysed the implications of sparsity within FMHSA to uncover whether we need sparse data-formats. (2) We used this analysis in order to propose the span-specialisation optimisation to reduce the amount of redundant work in FSMHSA.

Experimental Setup We implemented all our kernels in CUDA, using cuda-toolkit 11.6. All our experiments ran on A100 GPUs with 80GB of memory using single precision.

6.1 Evaluation

Analyzing the implication of sparsity. To understand how to exploit sparsity in FSMHSA, we conducted a motivational study. We instrumented our implementation in listing 1 counting the number of non-zero values computed in each $S_{i,j}$ block for the blocked pattern. The sparsity in this matrix is translated into the rest of the kernel, particularly the computation of $P_{i,j} \times O_{i,j}$. We analyzed the sparsity across iterations of j for thread-blocks in the top, middle, and end of the attention matrix. Fig 3 demonstrates our results. We get the important finding that different thread blocks experience non-zero computation at different times, commensurate with the sparse pattern they operate on. For both the dense patterns, the thread-blocks at the top will experience non-zero compute early, and towards the end they will experience non-zero compute late. This indicates that there is room to specialise the j iterations of each thread-block to reduce redundant compute at little to no cost. We uncover this next.

Span-specialisation. Based on the observation above, we attempted to apply span-specialisation to reduce the amount of redundant compute. This optimization first finds the starting and ending column indices of non-zero values across all the threads in a thread-block. Since the exact pattern is available just-in-time we can leverage the pattern information to compute these indices per thread-block before compiling and executing such a kernel. Doing so will prevent doing redundant work at little to no cost since the threads within a warp will do identical work, thus not deviating in control flow. Though we cut down the compute, our implementation nevertheless slows down the kernel substantially. This is demonstrated in figure 4. Across sparsity levels greater than 5%, our sparsity aware fusion is slower than its dense counterpart, linearly increasing in run-time until complete density. The cause of this is poor thread-access patterns to the Values matrix. As thread-blocks iterate across the j dimension, they read identical non-zero values from the Values matrix, thereby enhancing reuse in L2. However, by specialising the span of each thread-block, different thread-blocks read different chunks of the Values matrix, reducing reuse and polluting L2 caches, increasing the eviction rate and HBM memory accesses. We leave it to future work to understand how to mitigate this issue.

7 FUTURE WORK

First, our current implementation overuse shared memory. To reduce pressure on this data-store, we would like to explore how to use orthogonal memory stores, such as registers to accelerate these kernels.

Second, we explore span specialisation in the context of polygonal patterns. However, patterns with strides have spans across the entire attention matrix, resulting in the entire attention matrix computed. We wish to explore novel tiling strategies that help to mitigate this issue for this particular pattern.

Third, we wish to incorporate optimizations uncovered by flash-attention 2 for further improvement. Flash-attention 2 applies to only dense matrix-multiplication and it would be an interesting challenge to adapt it to the sparse case.

8 CONCLUSION

In this project, we have explored how to implement optimised sparse fused-multi-head-self-attention kernels. We have uncovered that the diverse set of patterns requires different optimizations, particularly polygonal patterns (blocked and windowed) against strided patterns. Moreover, we have uncovered that naive optimizations like span specialisations cause undue performance penalities due to irregular thread-access patterns across thread-blocks. This study has been insightful and has guided us on potential future research directions.

REFERENCES

[1] [n. d.]. cuBLAS — developer.nvidia.com. https://developer.nvidia.com/cublas.
[2] [n. d.]. cuSPARSE — developer.nvidia.com. https://developer.nvidia.com/cusparse.
[3] Anthropic. 2023. *Introducing 100K Context Windows.* https://www.anthropic.com/index/100k-context-windows
[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL]
[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165
[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End

Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[7] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating Long Sequences with Sparse Transformers. *CoRR* abs/1904.10509 (2019). arXiv:1904.10509 http://arxiv.org/abs/1904.10509

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG]

[9] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. arXiv:2006.10901 [cs.LG]

[10] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. *CoRR* abs/2001.04451 (2020). arXiv:2001.04451 https://arxiv.org/abs/2001.04451

[11] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[12] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen-tau Yih, Sinong Wang, and Jie Tang. 2019. Blockwise Self-Attention for Long Document Understanding. *CoRR* abs/1911.02972 (2019). arXiv:1911.02972 http://arxiv.org/abs/1911.02972

[13] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[14] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. Efficient Transformers: A Survey. *CoRR* abs/2009.06732 (2020). arXiv:2009.06732 https://arxiv.org/abs/2009.06732

[15] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009